Model-Based Testing

Pieter Koopman
pieter@cs.ru.nl

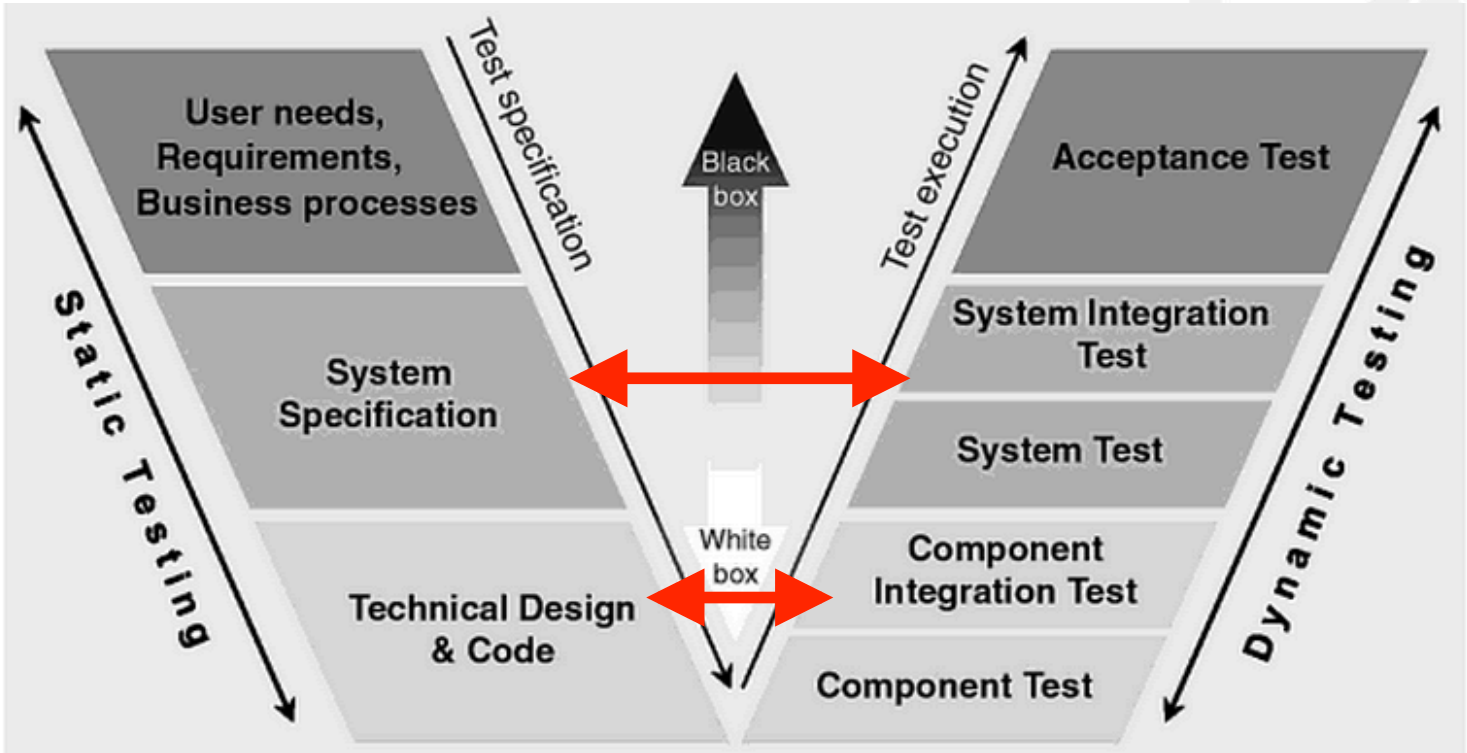
Model-Based System Development
Institute for Computing and Information Sciences (ICIS)
Radboud University Nijmegen
The Netherlands

model-based testing

- MBT is a special kind of automatic testing:
  1. test tool **generates test cases** based on model
     - usually on-the-fly
  2. generated tests are **executed automatically**
  3. test tool **gives verdict**
     - issue found, pass, proof

place in the software process

- MBT relates properties/specification to code



advantages of Model-Based Testing

- automatic test execution is cheap, fast, and reliable
  - more/better testing by changing parameter
- no **development** of test scripts
  - tests are dynamically generated from the model
- no **maintenance** of test scripts
  - maintain one model instead of set of test scripts
  - especially in agile development the system changes often
- there is a model that has a tested relation with the system
  - the model gives much more and better information than test scripts
  - development of the model is an useful activity on its own
- model can be used for other purposes
  - e.g. simulation, validation, shrinking

there are two flavours of Model-Based Testing

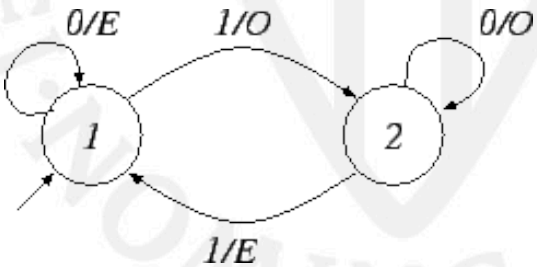
❑ property-based testing

$$\forall x \in R \,.\, x \geq 0 \Rightarrow (\sqrt{x})^2 = x$$

- ➢model is property of one function / method
  - e.g. the square of the square root of a number is that number

- ➢test tool tries to falsify this property by
1. generating test cases
2. evaluating the property for these arguments
3. give verdict based on test results

❑ state based conformance testing



- ➢model is a (extended) state machine
- ➢tested system behaves as state machine
- ➢test tool checks conformance of traces
  - sequence of inputs/outputs

property based testing

- application area:
  - everything determined by one input
  - this input can be arbitrary large and complex
    - argument of square root function
    - candidate list and all ballots of an election

property based testing

- based on general properties of methods, functions, systems
  - pension should increase if customer pays longer
- we use our own MBT-tool: **Gast**
  - distinguishing features:
  - systematic generation of test data generation
    - allows proof by exhaustive testing
  - generic (polytypic) generation of test data
    - systematic enumeration of values for any type can be derived by compiler
  - combines property based testing and state based conformance testing

simple examples of property bases testing

- the absolute value of each integers is greater or equal to

  $\forall n \in Z . |n| \geq 0$

- unit testing: check property for a number of values
  - abs (0) == 0
  - abs (-1) == 1
  - abs (10000) == 10000
  - abs (-100000) == 100000
- model based testing: state property in suitable form

```
pAbs :: Int -> Bool
pAbs n = abs n >= 0
```

- test result: ***Issue found*** *after 5 tests: -2147483648*
  - this is a consequence of 32-bit arithmetic

applications of property based testing

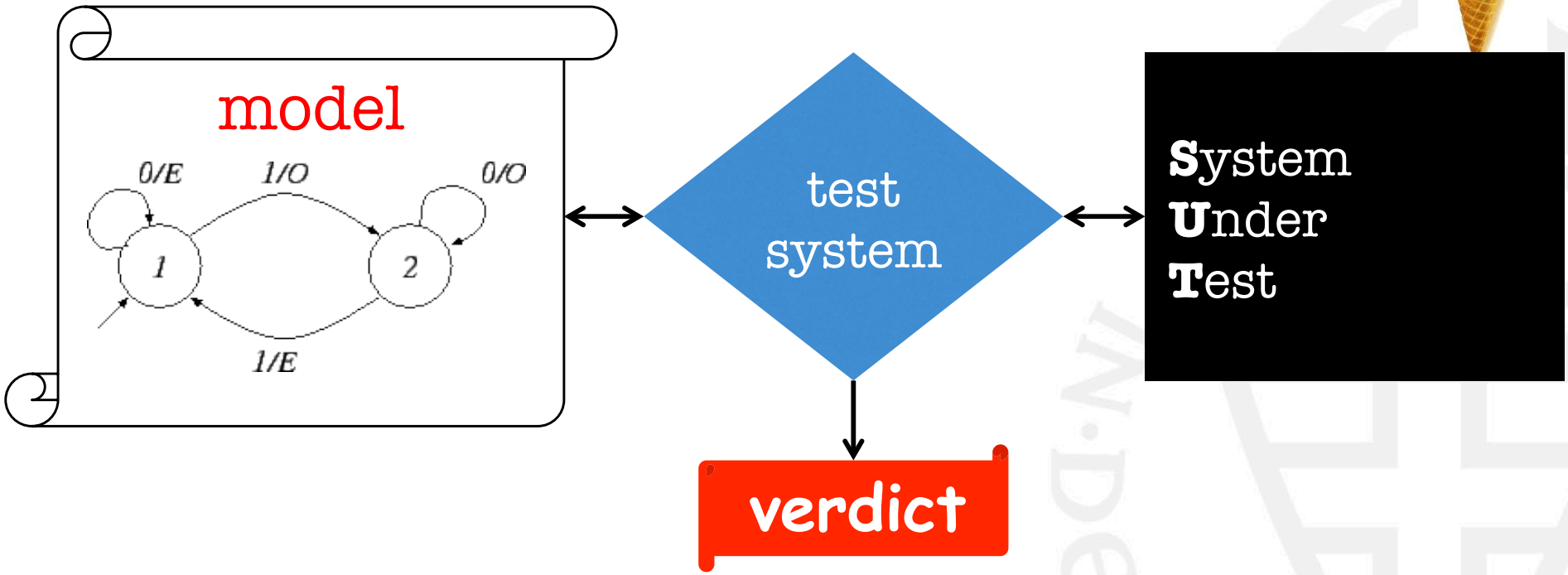
- Scottish elections
  - input: candidates, number to be elected, pile of ballots
  - output: elected persons (according to complex STV rules)
  - results:
    - developing the model revealed flaws in the election law
    - many errors found in second System Under Test
    - automatic test generation is even effective as hand crafted test cases

- claim computation of insurance company
  - input: data of person
  - output: claim rights
  - results:
    - serious errors found in a system that was in production for many years

state based conformance testing

- application area:
  - systems with a history (state)
  - vending machines, controllers, protocols, ..

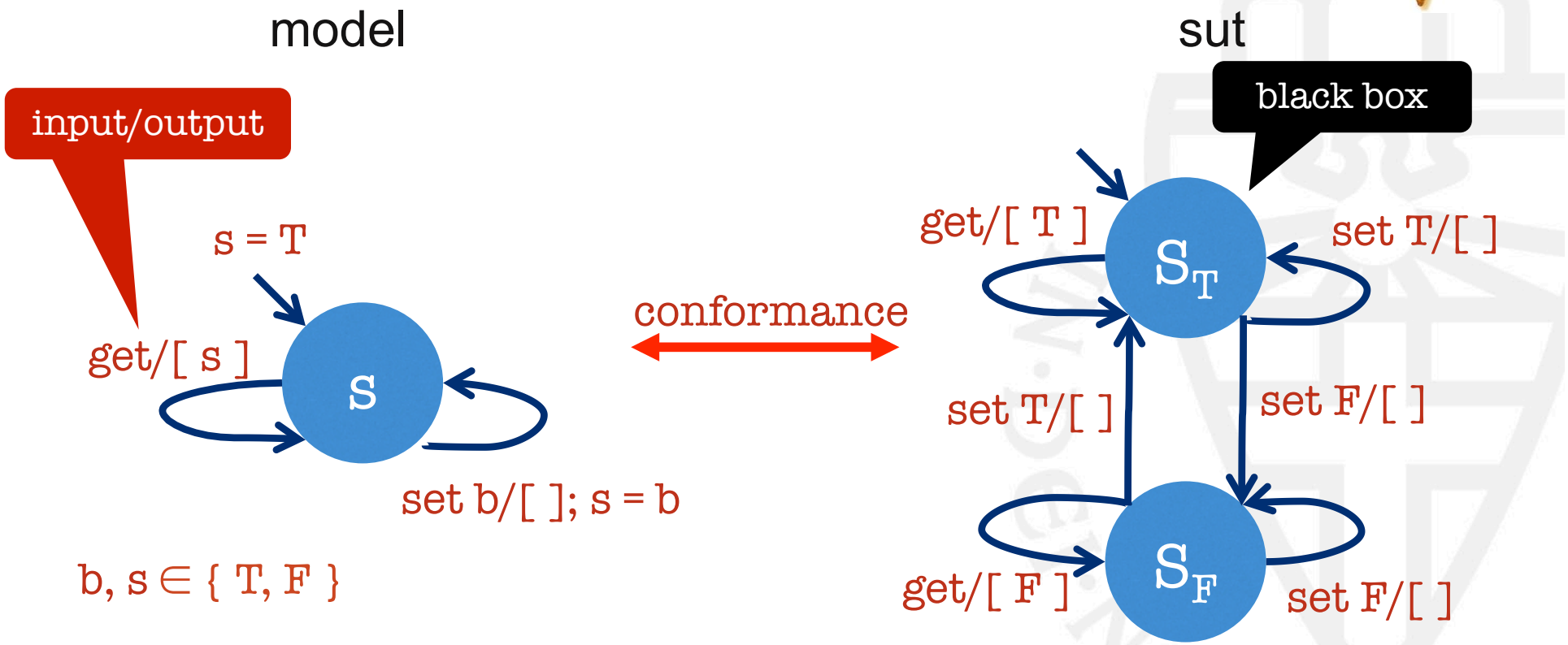
state based conformance testing



- try to falsify conformance by executing traces

1. select input allowed by specification
   - random walk or based on some test goal
2. apply input to the system under test (SUT)
3. check if output of SUT is allowed by the specification

state based conformance testing



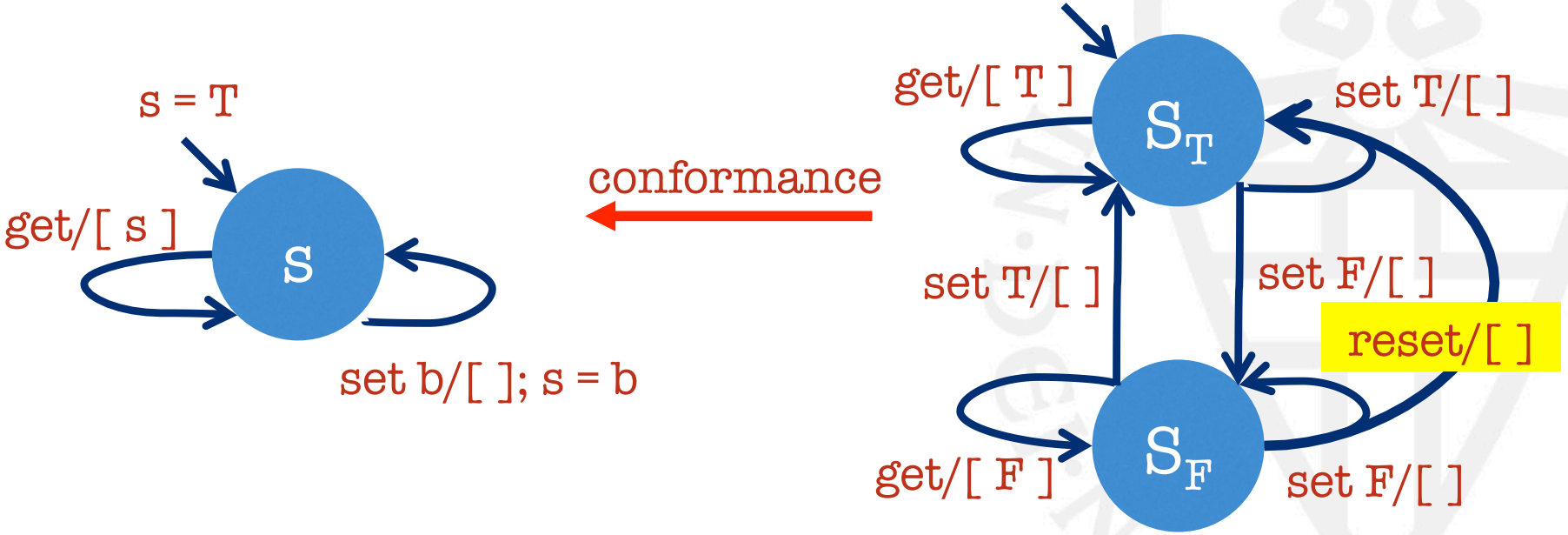
conformance: both machines have the same traces

since these are FSMs testing can prove conformance

state based conformance testing

model

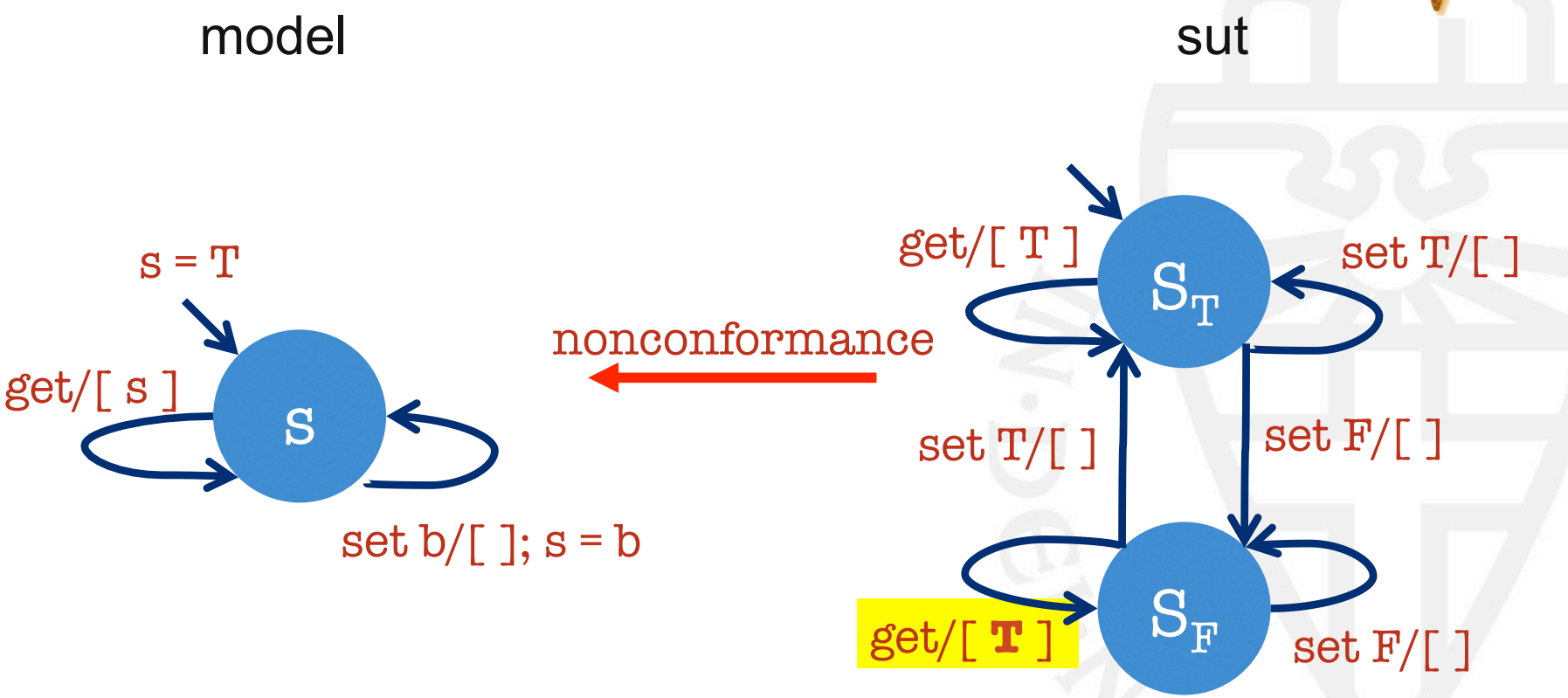
sut



reset is not part of the specification

partial specifications are fine

state based conformance testing



a trace showing nonconformance:

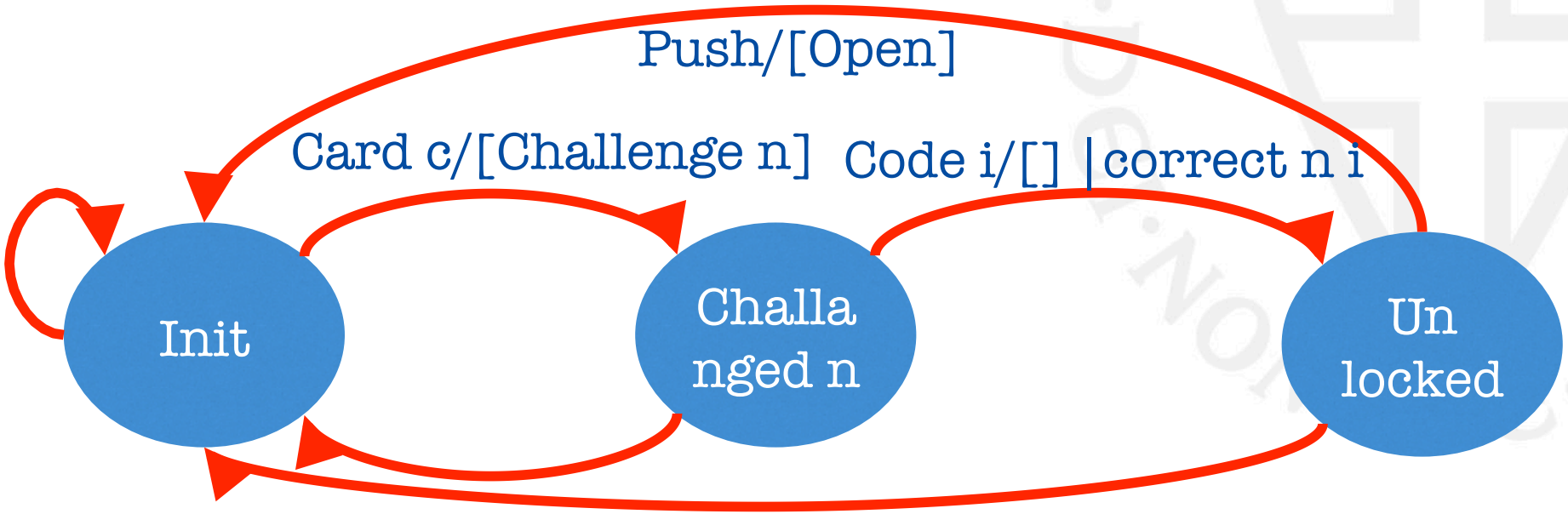
model allows only F

```
get/[T]; set F/[]; get/[T]
```

example: system to enter a building

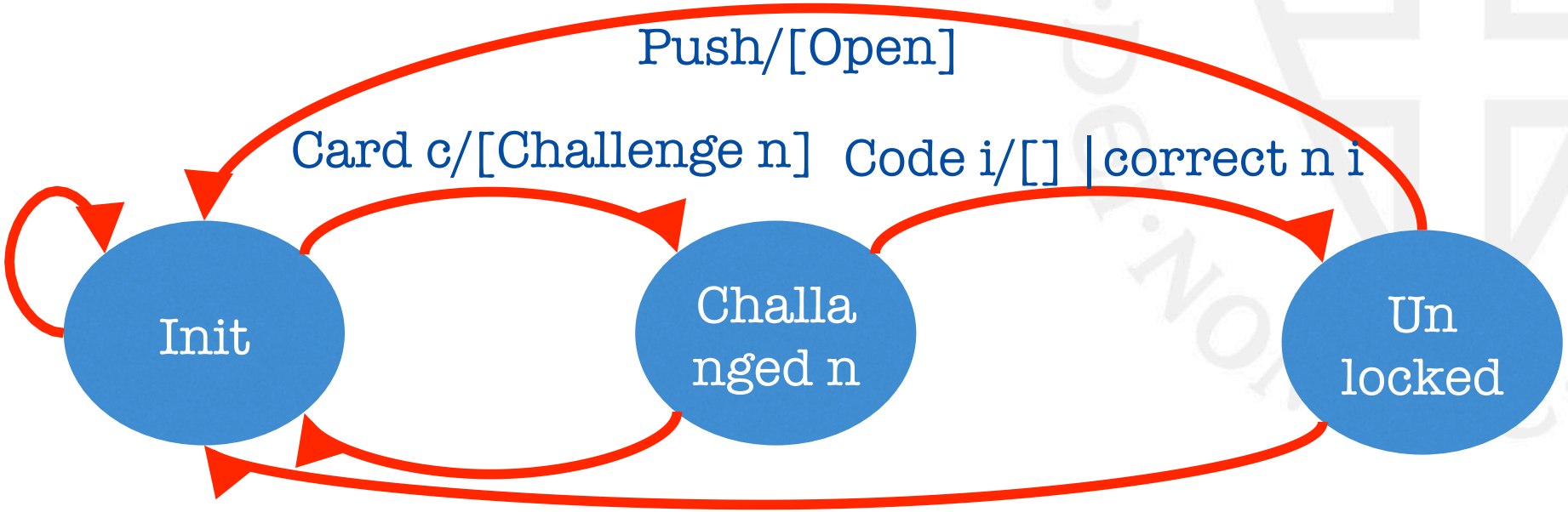
1. present card with valid number
2. get a challenge
3. enter the code
4. push the door to open

❑ in all other situation go to initial state



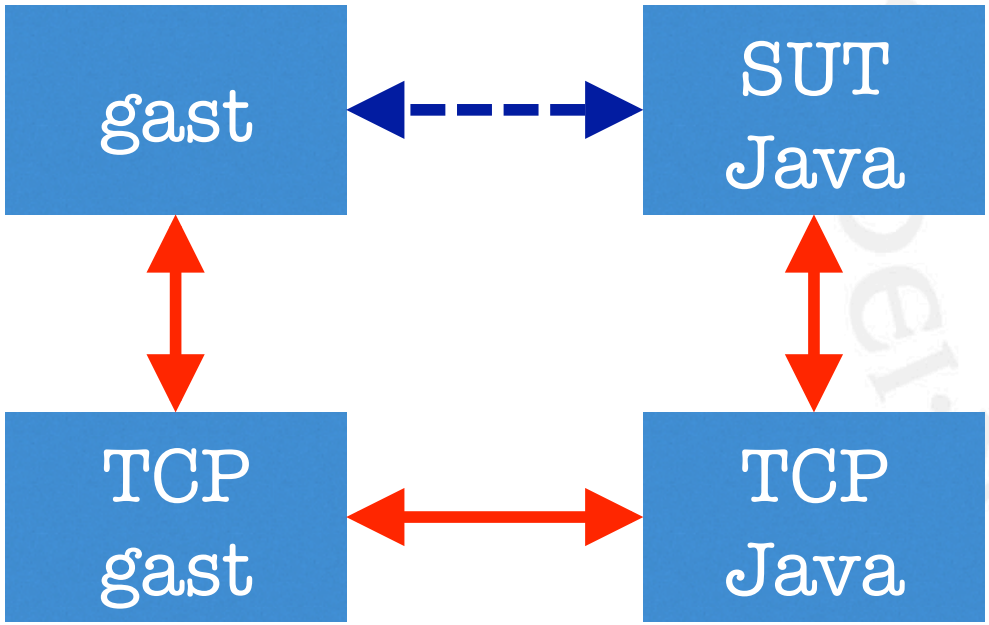
example: system to enter a building

```
spec :: State Input -> [Trans Output State]
spec Init (Card p) = [ Ft \i . case i of
                                 [Challenge c] = [Challenged c]
                                 [ ]           = [Init]
                                 _             = [ ] ]
spec (Challenged n) (Code i) = [Pt [ ] if (codeCorrect n i) Unlocked Init]
spec Unlocked       Push     = [Pt [Open] Init]
spec s              i        = [Pt [ ] Init]
```



test setup

- SUT is program in Java
- communication via TCP



trace showing an issue

SpecificationStates Input -> ObservedOutput

1: [Init] (Card 101) -> [Challenge 16]
2: [Challenged 16] (Code 8) -> [ ]
3: [Init] (Card 404) -> [ ]
4: [Init] (Card 202) -> [ ]
5: [Init] (Card 202) -> [ ]
6: [Init] (Card 202) -> [ ]
7: [Init] (Card 303) -> [Challenge 20]
8: [Challenged 20] (Code 0) -> [ ]
9: [Unlocked] Push -> [ ]

Allowed outputs and target states: [Pt [Opened] Init]

Issue found in path 5, in total 409 transitions.

general state machines as model in MBT

- we need arbitrary data types in state and transitions
  - the values of inputs and outputs are relevant for behaviour
  - to model correctness we need also arbitrary types in the state
  - the model can still be a huge abstraction of the system
    - hence the model is insufficient for model based system development
- we need nondeterministic models,
  even for deterministic systems
  - often the model has only partial information about the real world
  - model covers transitions corresponding to the missing information
- we need tooling to check and animate models
  - models are complex artefacts (like requirements and software)
- Gast offers this out of the box

applications of conformance testing

- embedded controller
  - issue in software generated from model that is proven to be correct
- Java card
- Dutch biometric passport
- printer controllers (Oce)
- Scottish elections, …

- Observations
  - this works very well
  - rule of thumb for script based testing:
    60% of the issues is caused by wrong test scripts
  - in MBT about 5% of the issues is due to modelling issues,
    checking the models will spot these issues before executing the tests

multi-purpose models

- making models is a valuable activity

multi-purpose models

- models developed for testing are valuable
  - even without a SUT
- developing a model reveals issues in specification
  - better than writing tests or the the application itself
- model can be tested to check properties
  - can this vending machine loose money, is it able to produce coffee
- one can prove properties about the model
- interactive simulation for validation
- model can be used to shrink traces with an issue
  - it is much easier to analyse an issue with a small trace
  - huge reductions (>90%) are common
- model is accurate specification of system
  - there is a tested relation between this model and the system

conclusion

- MBT adds an abstraction level to automated testing
  - tests are generated, executed and evaluated automatically
- the model has many advantages over test scripts
  - easy to generate more tests
  - model is much easier to maintain than set of scripts
  - model is a tested specification of the system: offers up to date documentation
  - one can validate the model before there is code
  - enables automatic shrinking of counterexamples
- it does require a different mental level
  - not all Dutch test engineers have the required capabilities